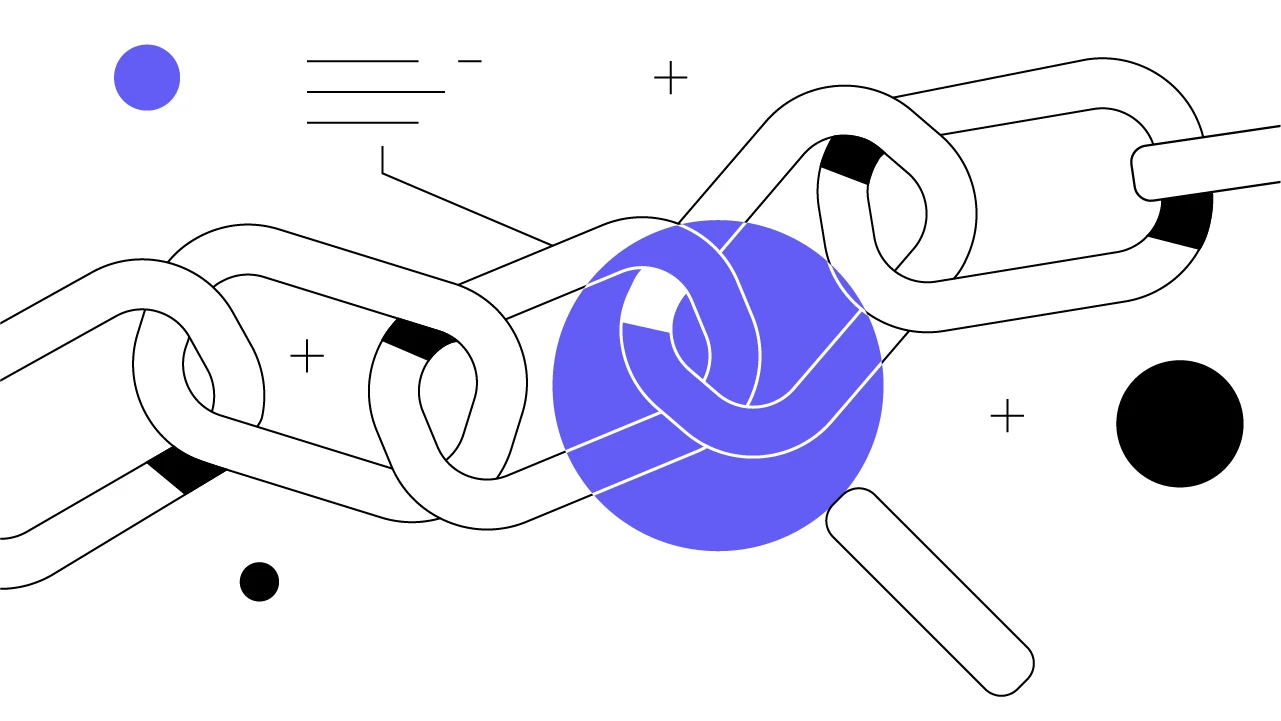
Project : Bitcoin Explorer: Part 1

# Introduction

Course: INFO7500 Cryptocurrency / Smart Contact FALL 2024

Team: Akshatha Patil, Sumanayana Konda, Ruthwik Bommenahalli Gowda

The Bitcoin Explorer project is a web application that provides real-time information about Bitcoin blocks. It demonstrates an end-to-end flow from user interface to database, including data ingestion from the Bitcoin network. This project is designed to showcase the integration of various technologies to create a functional Bitcoin block explorer.



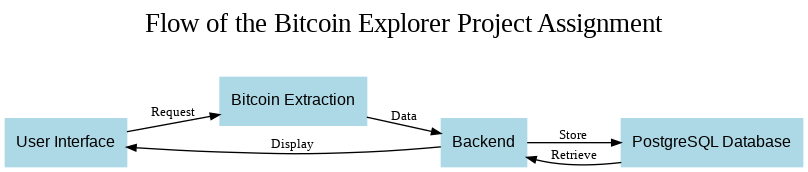
# Architecture

The project follows a three-tier architecture:

1. Frontend: React-based user interface
2. Backend: Node.js server with Express.js
3. Data Layer: PostgreSQL database and Rust-based Bitcoin data extractor

The application flow is as follows:

1. The Rust program extracts Bitcoin data from an API.
2. Extracted data is stored in the PostgreSQL database.
3. The Node.js backend serves this data through an API.
4. The React frontend fetches and displays the data.



# How to run the Application?

1. Setup PostgreSQL:
   * Install PostgreSQL
   * Create a database named 'postgres'
   * Create a table named 'bitcoin\_details' with columns 'height' (integer) and 'hash' (text)
2. Run the Rust Extractor:
   * Ensure Rust is installed
   * Navigate to the Rust project directory
   * Run cargo run
3. Start the Backend:
   * Navigate to the backend directory
   * Run npm install
   * Start the server with node index.js
4. Launch the Frontend:
   * Navigate to the frontend directory
   * Run npm install
   * Start the React app with npm start

# Bitcoin Extraction

Rust was chosen for Bitcoin data extraction due to its performance, safety, and robust ecosystem for concurrent programming. Here's a breakdown of the code and its functionality:

| async fn fetch\_bitcoin\_data(url: &str) -> Result<ApiResponse, ReqwestError> {  let response = reqwest::get(url).await?.json::<ApiResponse>().await?;  Ok(response)  }  async fn insert\_bitcoin\_data(client: &Client, details: &ApiResponse) -> Result<(), PgError> {  let height = details.height as i64;  client.execute(  "INSERT INTO bitcoin\_details (height, hash) VALUES ($1, $2)",  &[&height, &details.hash],  ).await?;  Ok(())  }  #[tokio::main]  async fn main() -> Result<(), Box<dyn std::error::Error>> {  // ... (connection setup)  match fetch\_bitcoin\_data(url).await {  Ok(details) => {  if let Err(e) = insert\_bitcoin\_data(&client, &details).await {  eprintln!("Error inserting data: {:?}", e);  } else {  println!("Data inserted successfully.");  }  },  Err(e) => {  eprintln!("Error fetching data: {:?}", e);  }  }  Ok(())  } |
| --- |

## How it works:

1. The fetch\_bitcoin\_data function uses the reqwest library to make an asynchronous HTTP GET request to the BlockCypher API and deserialize the JSON response into an ApiResponse struct.
2. insert\_bitcoin\_data takes the fetched data and inserts it into the PostgreSQL database using the tokio-postgres library.
3. The main function, marked with #[tokio::main], sets up the database connection and orchestrates the fetching and insertion process.

## Why Rust?

* Performance: Rust's zero-cost abstractions and efficient memory management make it ideal for high-performance data processing.
* Safety: Rust's ownership system and strict compile-time checks prevent common programming errors like null or dangling pointer references.
* Concurrency: Rust's async/await syntax, coupled with the tokio runtime, allows for efficient handling of concurrent operations, which is crucial for network requests and database interactions.
* Strong typing: Rust's type system ensures that data is correctly structured and handled throughout the application.

# Backend : NodeJS

Node.js was selected for the backend due to its event-driven, non-blocking I/O model, which is well-suited for building scalable network applications. Here's an analysis of the backend code:

| const express = require('express');  const cors = require('cors');  const { Pool } = require('pg');  const app = express();  app.use(cors());  const pool = new Pool({  host: 'localhost',  user: 'postgres',  password: 'postgres',  database: 'postgres',  port: 5432  });  app.get('/api/bitcoin-details', async (req, res) => {  try {  const result = await pool.query('SELECT \* FROM bitcoin\_details');  res.json(result.rows);  } catch (error) {  console.error('Error fetching data from PostgreSQL:', error);  res.status(500).send('Server error');  }  });  app.listen(PORT, () => {  console.log(`Server is running on http://localhost:${PORT}`);  }); |
| --- |

## How it works:

1. The Express.js framework is used to create a web server and define routes.
2. CORS middleware is applied to allow cross-origin requests, essential for separating frontend and backend.
3. A connection pool for PostgreSQL is created using the pg library, allowing for efficient database connections.
4. The /api/bitcoin-details endpoint queries the database for all Bitcoin details and returns them as JSON.
5. Error handling is implemented to catch and log any issues during database queries or server operations.

## Why Node.js?

* Asynchronous and Event-Driven: Node.js excels at handling multiple concurrent connections without the overhead of creating separate threads.
* JavaScript Ecosystem: The vast npm ecosystem provides a wealth of libraries and tools, speeding up development.
* JSON Handling: As a JavaScript runtime, Node.js has excellent native support for working with JSON, the primary data format for web APIs.
* Scalability: Node.js's non-blocking I/O model makes it easy to scale applications horizontally.

# Frontend : React

React.js was chosen for the frontend due to its component-based architecture and efficient rendering mechanism. Here's a detailed look at key parts of the frontend code:

| function App() {  const [bitcoinDetails, setBitcoinDetails] = useState([]);  const [isLoading, setIsLoading] = useState(true);  const [error, setError] = useState(null);  useEffect(() => {  fetch('http://localhost:3000/api/bitcoin-details')  .then(response => {  if (!response.ok) {  throw new Error('Network response was not ok');  }  return response.json();  })  .then(data => {  console.log("Fetched Bitcoin Data:", data);  setBitcoinDetails(data);  setIsLoading(false);  })  .catch(error => {  console.error('Error fetching data:', error);  setError(error.message);  setIsLoading(false);  });  }, []);  // ... (render logic)  } |
| --- |

## How it works:

1. The useState hook is used to manage component state for Bitcoin details, loading status, and errors.
2. useEffect is employed to fetch data from the backend API when the component mounts.
3. The fetched data is stored in state, and loading/error states are updated accordingly.
4. The component renders different UI elements based on the current state (loading, error, or data display).
5. Styled components are used to create a visually appealing and responsive design.

## Why React.js?

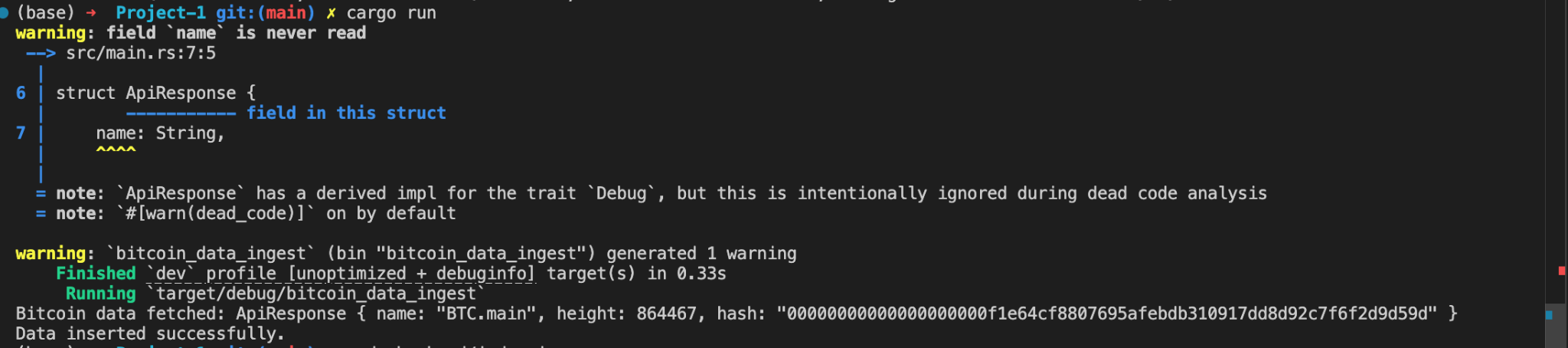
* Component-Based Architecture: React's component model allows for the creation of reusable UI elements, improving code organization and maintainability.
* Virtual DOM: React's virtual DOM optimizes rendering performance by minimizing actual DOM manipulations.
* Unidirectional Data Flow: React's state management and props system make it easier to track and debug data changes in the application.
* Large Ecosystem: The React ecosystem offers a wide range of libraries and tools for state management, routing, and UI components.
* Developer Experience: Features like hot reloading and excellent developer tools enhance the development process.

# Results

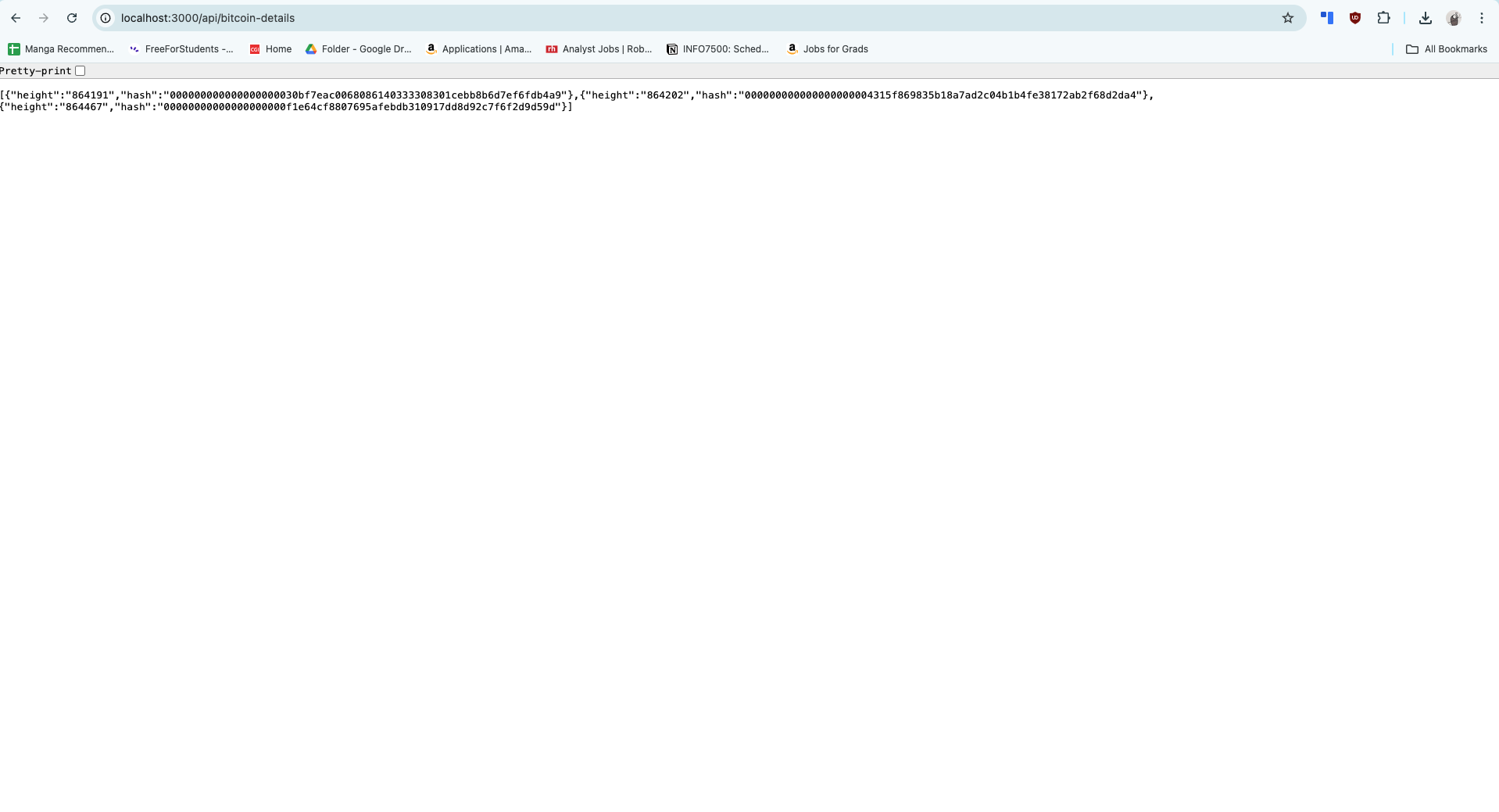
By leveraging these technologies - Rust for efficient data extraction, Node.js for a scalable backend, and React.js for a dynamic frontend - the Bitcoin Explorer project achieves a balance of performance, scalability, and user experience. This architecture allows for real-time data processing and display, crucial for a blockchain explorer application.

When running successfully, the application will display a table of Bitcoin block heights and their corresponding hashes. The data is fetched from the backend API and updated in real-time as new blocks are added to the Bitcoin blockchain.

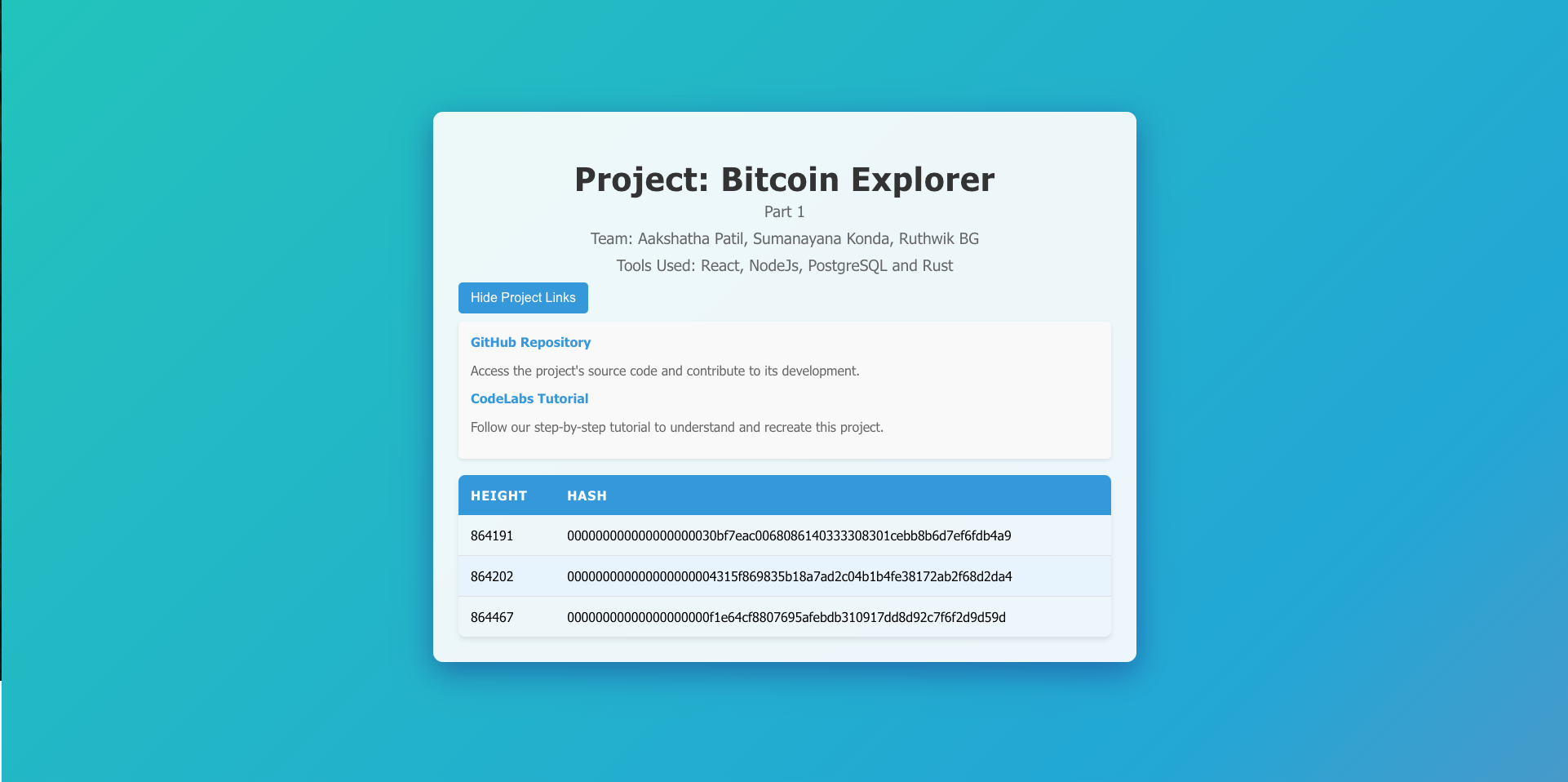
Bitcoin Extraction:

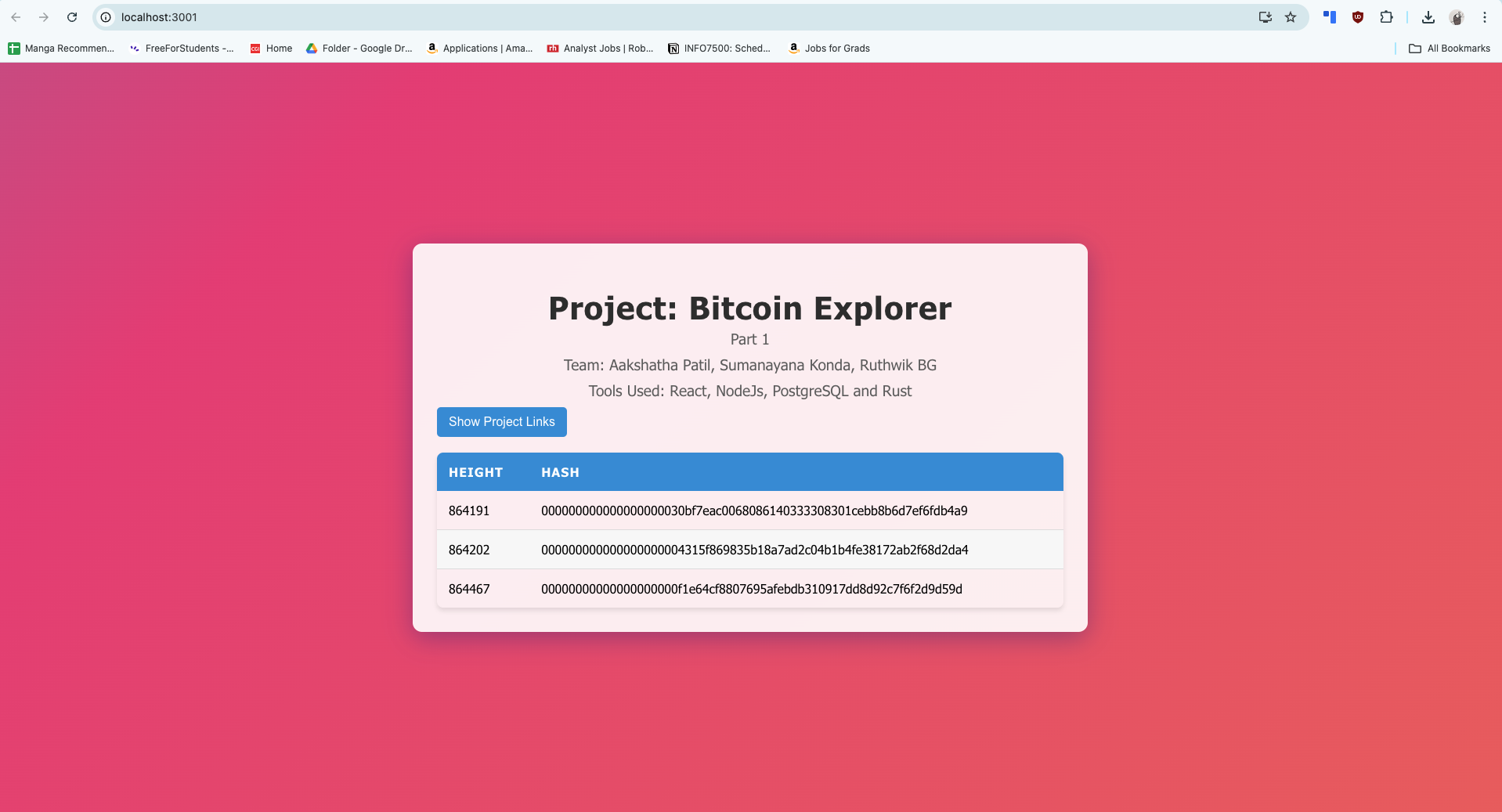


Backend:



Frontend:





# References

## 

1. React Documentation: <https://reactjs.org/docs/getting-started.html>
2. Node.js Documentation: <https://nodejs.org/en/docs/>
3. Rust Documentation: <https://doc.rust-lang.org/book/>
4. PostgreSQL Documentation: <https://www.postgresql.org/docs/>
5. BlockCypher API Documentation: <https://www.blockcypher.com/dev/bitcoin/>

### 